XSCHEME: An Object-oriented Scheme

Version 0.28

July 23, 1991

by
David Michael Betz
P.O. Box 144
Peterborough, NH 03458

(603) 924-4145 (home)

Copyright (c) 1991, by David Michael Betz
All Rights Reserved
Permission is granted for unrestricted non-commercial use

TABLE OF CONTENTS

TABLE OF CONTENTS.....2
INTRODUCTION.....3
A NOTE FROM THE AUTHOR.....4
EXPRESSIONS.....5
BINDING FORMS.....10
SEQUENCING.....11
DELAYED EVALUATION.....12
ITERATION.....13
DEFINITIONS.....14
LIST FUNCTIONS.....15
DESTRUCTIVE LIST FUNCTIONS.....17
SYMBOL FUNCTIONS.....18
VECTOR FUNCTIONS.....19
ARRAY FUNCTIONS.....20
CONVERSION FUNCTIONS.....21
TYPE PREDICATES.....22
EQUALITY PREDICATES.....25
ARITHMETIC FUNCTIONS.....26
NUMERIC COMPARISON FUNCTIONS.....29
BITWISE LOGICAL FUNCTIONS.....30
STRING FUNCTIONS.....31
STRING COMPARISON FUNCTIONS.....32
CHARACTER COMPARISON FUNCTIONS.....33
INPUT/OUTPUT FUNCTIONS.....34
OUTPUT CONTROL FUNCTIONS.....36
FILE I/O FUNCTIONS.....37
CONTROL FEATURES.....39
ENVIRONMENT FUNCTIONS.....40
UTILITY FUNCTIONS.....41
SYSTEM FUNCTIONS.....42
OBJECT REPRESENTATIONS.....43

XScheme is an implementation of the Scheme programming language with extensions to support object-oriented programming.

There are currently implementations of XScheme running on the IBM-PC and clones under MS-DOS, on the Macintosh, the Atari-ST and the Amiga. It is completely written in the programming language 'C' and is easily extended with user written built-in functions and classes. It is available in source form to non-commercial users.

This document is a brief description of XScheme. XScheme follows the "Revised^3 Report on the Algorithmic Language Scheme". It assumes some knowledge of Scheme or LISP and some understanding of the concepts of object-oriented programming.

I recommend the book "Structure and Interpretation of Computer Programs" by Harold Abelson and Gerald Jay Sussman and published by The MIT Press and the McGraw-Hill Book Company for learning Scheme (and programming in general). You might also find "The Scheme Programming Language" by R. Kent Dybvig and "The Little Lisper" by Daniel P. Friedman and Matthias Felleisen to be helpful.

A NOTE FROM THE AUTHOR

If you have any problems with XScheme, feel free to contact me for help or advice. Please remember that since XScheme is available in source form in a high level language, many users have been making versions available on a variety of machines. If you call to report a problem with a specific version, I may not be able to help you if that version runs on a machine to which I don't have access. Please have the version number of the version that you are running readily accessible before calling me.

If you find a bug in XScheme, first try to fix the bug yourself using the source code provided. If you are successful in fixing the bug, send the bug report along with the fix to me. If you don't have access to a C compiler or are unable to fix a bug, please send the bug report to me and I'll try to fix it.

Any suggestions for improvements will be welcomed. Feel free to extend the language in whatever way suits your needs. However, PLEASE DO NOT RELEASE ENHANCED VERSIONS WITHOUT CHECKING WITH ME FIRST!! I would like to be the clearing house for new features added to XScheme. If you want to add features for your own personal use, go ahead. But, if you want to distribute your enhanced version, contact me first.

EXPRESSIONS

<variable>

An expression consisting of a variable is a variable reference. The value of the variable reference is the value stored in the location to which the variable is bound. It is an error to reference an unbound variable.

(QUOTE <datum>)
'<datum>

(quote <datum>) evaluates to <datum>. <Datum> may be any external representation of a Scheme object. This notation is used to include literal constants in Scheme code. (quote <datum>) may be abbreviated as '<datum>. The two notations are equivalent in all respects.

<constant>

Numeric constants, string constants, character constants, and boolean constants evaluate "to themselves"; they need not be quoted.

(<operator> <operand>...)

A procedure call is written by simply enclosing in parentheses expressions for the procedure to be called and the arguments to be passed to it. The operator and operand expressions are evaluated and the resulting procedure is passed the resulting arguments.

(<object> <selector> <operand>...)

A message sending form is written by enclosing in parentheses expressions for the receiving object, the message selector, and the arguments to be passed to the method. The receiver, selector, and argument expressions are evaluated, the message selector is used to select an appropriate method to handle the message, and the resulting

method is passed the resulting arguments.

(LAMBDA <formals> <body>)

<Formals> should be a formal argument list as described below, and <body> should be a sequence of one or more expressions. A lambda expression evaluates to a procedure. The environment in effect when the lambda expression is evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the lambda expression was evaluated will be extended by binding the variables in the formal argument list to fresh locations, the corresponding actual argument values will be stored in those locations, and the expressions in the body of the lambda expression will be evaluated sequentially in the extended environment. The result of the last expression in the body will be returned as the result of the procedure call.

<Formals> should have the following form:

```
    (<var>... [#!OPTIONAL <ovar>...] [. <rvar>])
  or
    (<var>... [#!OPTIONAL <ovar>...] [#!REST <rvar>])

  where:

    <var>    is a required argument
    <ovar>   is an optional argument
    <rvar>   is a "rest" argument
```

There are three parts to a <formals> list. The first lists the required arguments of the procedure. All calls to the procedure must supply values for each of the required arguments. The second part lists the optional arguments of the procedure. An optional argument may be supplied in a call or omitted. If it is omitted, a special value is given to the argument that satisfies the default-object?

predicate. This provides a way to test to see if an optional argument was provided in a call or omitted. The last part of the <formals> list gives the "rest" argument. This argument will be bound to the rest of the list of arguments supplied to a call after the required and optional arguments have been removed.

(IF <test> <consequent> [<alternate>])

An if expression is evaluated as follows: first, <test> is evaluated. If it yields a true value, then <consequent> is evaluated and its value is returned. Otherwise, <alternate> is evaluated and its value is returned. If <test> yields a false value and no <alternate> is specified, then the result of the expression is unspecified.

(ACCESS <variable> <env>)

<Env> is evaluated producing an environment. The result is the value of <variable> in this environment.

(SET! <variable> <expression>)

<Expression> is evaluated, and the resulting value is stored in the location to which <variable> is bound. <Variable> must be bound in some region or at the top level. The result of the set! expression is unspecified.

(SET! (ACCESS <variable> <env>) <value>)

<Env> is evaluated producing an environment. <Value> is evaluated and the resulting value is stored as the value of <variable> in this environment. The result of the set! expression is unspecified.

(COND <clause>...)

Each clause should be of the form

(<test> <expression>...)

where <test> is any expression. The last <clause> may be an "else clause," which has the form

(ELSE <expression>...)

A cond expression is evaluated by evaluating the <test> expressions of successive <clause>s in order until one of them evaluates to a true value. When a <test> evaluates to a true value, then the remaining <expression>s in its <clause> are evaluated in order, and the result of the last <expression> in the <clause> is returned as the result of the entire cond expression. If the selected <clause> contains only the <test> and no <expression>s, then the value of the <test> is returned as the result. If all <test>s evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its <expression>s are evaluated, and the value of the last one is returned.

(AND <test>...)

The <test> expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then #t is returned.

(OR <test>...)

The <test> expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then #f is returned.

BINDING FORMS

(LET [<name>] <bindings> <body>)

<Bindings> should have the form

((<variable> <init>)...)

where each <init> is an expression, and <body> should be a sequence of one or more expressions. The <init>s are evaluated in the current envirnoment, the <variable>s are bound to fresh locations holding the results, the <body> is evaluated in the extended environment, and the value of the last expression of <body> is returned. Each binding of a <variable> has <body> as its region.

If a name is supplied, a procedure that takes the bound variables as its arguments and has the body of the LET as its body is bound to that name.

(LET* <bindings> <body>)

Same as LET except that the bindings are done sequentially from left to right and the bindings to the left are visible while evaluating the initialization expressions for each variable.

(LETREC <bindings> <body>)

<Bindings> should have the form

((<variable> <init>)...)

and <body> should be a sequence of one or more expressions. The <variable>s are bound to fresh locations holding undefined values; the <init>s are evaluated in the resulting environment; each <variable> is assigned to the result of the corresponding <init>; the <body> is evaluated in the resulting environment; and the value of the last expression in <body> is returned. Each binding of a <variable> has the entire letrec expression as its region, making it possible to define mutually recursive procedures. One restriction of letrec is very important: it must be possible to evaluate each <init> without referring to the value of any <variable>. If this restriction is violated, then the effect is undefined, and an error may be signalled during evaluation of the <init>s. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of letrec, all the <init>s are lambda expressions and the restriction is satisfied automatically.

SEQUENCING

(BEGIN <expression>...)
(SEQUENCE <expression>...)

The <expression>s are evaluated sequentially from left to right, and the value of the last <expression> is returned. This expression type is used to sequence side effects such as input and output.

DELAYED EVALUATION

(CONS-STREAM expr1 expr2)

Create a cons stream whose head is expr1 (which is evaluated immediately) and whose tail is expr2 (whose evaluation is delayed until TAIL is called). To use CONS-STREAM, enter the following access procedures:

```
(define head car)
(define (tail stream) (force (cdr stream)))
```

(DELAY <expression>)

Evaluating this expression creates a "promise" to evaluate <expression> at a later time.

(FORCE promise)

Applying FORCE to a promise generated by DELAY requests that the promise produce the value of the expression passed to DELAY. The first time a promise is FORCEed, the DELAY expression is evaluated and the value stored. On subsequent calls to FORCE with the same promise, the saved value is returned.

ITERATION

(WHILE <test> <expression>...)

While is an iteration construct. Each iteration begins by evaluating <test>; if the result is false, then the loop terminates and the value of <test> is returned as the value of the while expression. If <test> evaluates

to a true value, then the <expression>s are evaluated in order for effect and the next iteration begins.

DEFINITIONS

(DEFINE <variable> <expression>)

Define a variable and give it an initial value.

(DEFINE (<variable> <formals>) <body>)

Define a procedure.

LIST FUNCTIONS

(CONS expr1 expr2)

Create a new pair whose car is expr1 and whose cdr is expr2.

(CAR pair)

Extract the car of a pair.

(CDR pair)

Extract the cdr of a pair.

(CxxR pair)
(CxxxR pair)
(CxxxxR pair)

These functions are short for combinations of CAR and CDR. Each 'x' is stands for either 'A' or 'D'. An 'A' stands for the CAR function and a 'D' stands for the CDR function. For instance, (CADR x) is the same as (CAR (CDR x)).

(LIST expr...)

Create a list whose elements are the arguments to the function. This function can take an arbitrary number of arguments. Passing no arguments results in the empty list.

(LIST* expr...)

Create a list whose elements are the arguments to the function except that the last argument is used as the tail of the list. This means that the call (LIST* 1 2 3) produce the result (1 2 . 3).

(APPEND list...)

Append lists to form a single list. This function takes an arbitrary number of arguments. Passing no arguments results in the empty list.

(REVERSE list)

Create a list whose elements are the same as the argument except in reverse order.

(LAST-PAIR list)

Return the last pair in a list.

(LENGTH list)

Compute the length of a list.

(MEMBER expr list)
(MEMV expr list)
(MEMQ expr list)

Find an element in a list. Each of these functions searches the list looking for an element that matches expr. If a matching element is found, the remainder of the list starting with that element is returned. If no matching element is found, the empty list is returned. The functions differ in the test used to determine if an element matches expr. The MEMBER function uses EQUAL?, the MEMV function uses EQV? and the MEMQ function uses EQ?.

(ASSOC expr alist)
(ASSV expr alist)
(ASSQ expr alist)

Find an entry in an association list. An association list is a list of pairs. The car of each pair is the key and the cdr is the value. These functions search an association list for a pair whose key matches expr. If a matching pair is found, it is returned. Otherwise, the empty list is returned. The functions differ in the test used to determine if a key matches expr. The ASSOC function uses EQUAL?, the ASSV function uses EQV? and the ASSQ function uses EQ?.

(LIST-REF list n)

Return the nth element of a list (zero based).

(LIST-TAIL list n)

Return the sublist obtained by removing the first n elements of list.

DESTRUCTIVE LIST FUNCTIONS

(SET-CAR! pair expr)

Set the car of a pair to expr. The value returned by this procedure is unspecified.

(SET-CDR! pair expr)

Set the cdr of a pair to expr. The value returned by this procedure is unspecified.

SYMBOL FUNCTIONS

(BOUND? sym)

Returns #t if a global value is bound to the symbol and #f otherwise.

(SYMBOL-VALUE sym)

Get the global value of a symbol.

(SET-SYMBOL-VALUE! sym expr)

Set the global value of a symbol. The value returned by this procedure is unspecified.

(SYMBOL-PLIST sym)

Get the property list associated with a symbol.

(SET-SYMBOL-PLIST! sym plist)

Set the property list associate with a symbol. The value returned by this procedure is unspecified.

(GENSYM [sym|str|num])

Generate a new, uninterned symbol. The print name of the symbol will consist of a prefix with a number appended. The initial prefix is "G" and the initial number is 1. If a symbol is specified as an argument, the prefix is set to the print name of that symbol. If a string is specified, the prefix is set to that string. If a number is specified, the numeric suffix is set to that number. After the symbol is generated, the number is incremented so subsequent calls to GENSYM will generate numbers in sequence.

(GET sym prop)

Get the value of a property of a symbol. The prop argument is a symbol that is the property name. If a property with that name exists on the symbols property list, the value of the property is returned. Otherwise, the empty list is returned.

(PUT sym prop expr)

Set the value of a property of a symbol. The prop argument is a symbol that is the property name. The

property/value combination is added to the property list of the symbol.

VECTOR FUNCTIONS

(VECTOR expr...)

Create a vector whose elements are the arguments to the function. This function can take an arbitrary number of arguments. Passing no arguments results in a zero length vector.

(MAKE-VECTOR len)

Make a vector of the specified length.

(VECTOR-LENGTH vect)

Get the length of a vector.

(VECTOR-REF vect n)

Return the nth element of a vector (zero based).

(VECTOR-SET! vect n expr)

Set the nth element of a vector (zero based).

ARRAY FUNCTIONS

(MAKE-ARRAY d1 d2...)

Make an array (vector of vectors) with the specified dimensions. At least one dimension must be specified.

(ARRAY-REF array s1 s2...)

Get an array element. The sn arguments are integer subscripts (zero based).

(ARRAY-SET! array s1 s2... expr)

Set an array element. The sn arguments are integer subscripts (zero based).

CONVERSION FUNCTIONS

(SYMBOL->STRING sym)

Convert a symbol to a string. Returns the print name of the symbol as a string.

(STRING->SYMBOL str)

Convert a string to a symbol. Returns a symbol with the string as its print name. This can either be a new symbol or an existing one with the same print name.

(VECTOR->LIST vect)

Convert a vector to a list. Returns a list of the elements of the vector.

(LIST->VECTOR list)

Convert a list to a vector. Returns a vector of the elements of the list.

(STRING->LIST str)

Convert a string to a list. Returns a list of the characters in the string.

(LIST->STRING list)

Convert a list of character to a string. Returns a string whose characters are the elements of the list.

(CHAR->INTEGER char)

Convert a character to an integer. Returns the ASCII code of the character as an integer.

(INTEGER->CHAR n)

Convert an integer ASCII code to a character. Returns the character whose ASCII code is the integer.

TYPE PREDICATE FUNCTIONS

(NOT expr)

Returns #t if the expression is #f and #t otherwise.

(NULL? expr)

Returns #t if the expression is the empty list and #f otherwise.

(ATOM? expr)

Returns #f if the expression is a pair and #t otherwise.

(LIST? expr)

Returns #t if the expression is either a pair or the empty list and #f otherwise.

(NUMBER? expr)

Returns #t if the expression is a number and #f otherwise.

(BOOLEAN? expr)

Returns #t if the expression is either #t or #f and #f otherwise.

(PAIR? expr)

Returns #t if the expression is a pair and #f otherwise.

(SYMBOL? expr)

    Returns #t if the expression is a symbol and #f
    otherwise.

(COMPLEX? expr)

    Returns #t if the expression is a complex number and #f
    otherwise.
    Note: Complex numbers are not yet supported by XScheme.

(REAL? expr)

    Returns #t if the expression is a real number and #f
    otherwise.

(RATIONAL? expr)

    Returns #t if the expression is a rational number and #f
    otherwise.
    Note: Rational numbers are not yet supported by
    XScheme.

(INTEGER? expr)

    Returns #t if the expression is an integer and #f
    otherwise.

(CHAR? expr)

    Returns #t if the expression is a character and #f
    otherwise.

(STRING? expr)

    Returns # if the expression is a string and #f
    otherwise.

(VECTOR? expr)

    Returns #t if the expression is a vector and #f
    otherwise.

(PROCEDURE? expr)

    Returns #t if the expression is a procedure (closure)

and #f otherwise.

(PORT? expr)

Returns #t if the expression is a port and #f otherwise.

(INPUT-PORT? expr)

Returns #t if the expression is an input port and #f otherwise.

(OUTPUT-PORT? expr)

Returns #t if the expression is an output port and #f otherwise.

(OBJECT? expr)

Returns #t if the expression is an object and #f otherwise.

(EOF-OBJECT? expr)

Returns #t if the expression is the object returned by READ upon detecting an end of file condition and #f otherwise.

(DEFAULT-OBJECT? expr)

Returns #t if the expression is the object passed as the default value of an optional parameter to a procedure when that parameter is omitted from a call and #f otherwise.

(ENVIRONMENT? x)

Returns #t if the expression is an environment and #f otherwise.

EQUALITY PREDICATES

(EQUAL? expr1 expr2)

Recursively compares two objects to determine if their components are the same and returns #t if they are the same and #f otherwise.

(EQV? expr1 expr2)

Compares two objects to determine if they are the same object. Returns #t if they are the same and #f otherwise. This function does not compare the elements of lists, vectors or strings but will compare all types of numbers.

(EQ? expr1 expr2)

Compares two objects to determine if they are the same object. Returns #t if they are the same and #f otherwise. This function performs a low level address compare on two objects and may return #f for objects

that appear on the surface to be the same. This is because the objects are not stored uniquely in memory. For instance, numbers may appear to be equal, but EQ? will return #f when comparing them if they are stored at different addresses. The advantage of this function is that it is faster than the others. Symbols are guaranteed to compare correctly, so EQ? can safely be used to compare them.

ARITHMETIC FUNCTIONS

(ZERO? n)

Returns #t if the number is zero and #f otherwise.

(POSITIVE? n)

Returns #t if the number is positive and #f otherwise.

(NEGATIVE? n)

Returns #t if the number is negative and #f otherwise.

(ODD? n)

Returns #t if the integer is odd and #f otherwise.

(EVEN? n)

    Returns #t if the integer is even and #f otherwise.

(EXACT? n)

    Returns #t if the number is exact and #f otherwise.
    Note: This function always returns #f in XScheme since
    exact numbers are not yet supported.

(INEXACT? n)

    Returns #t if the number is inexact and #f otherwise.
    Note: This function always returns #t in XScheme since
    exact numbers are not yet supported.

(TRUNCATE n)

    Truncates a number to an integer and returns the
    resulting value.

(FLOOR n)

    Returns the largest integer not larger than n.

(CEILING n)

    Returns the smallest integer not smaller than n.

(ROUND n)

    Returns the closest integer to n, rounding to even when
    n is halfway between two integers.

(1+ n)

    Returns the result of adding one to the number.

(-1+ n)

    Returns the result of subtracting one from the number.

(ABS n)

    Returns the absolute value of the number.

(GCD n1 n2)

    Returns the greatest common divisor of the two numbers.

(RANDOM n)

Returns a random number between zero and n-1 (n must be an integer).

(+ n1 n2...)

Returns the sum of the numbers.

(- n)

Negates the number and returns the resulting value.

(- n1 n2...)

Subtracts each remaining number from n1 and returns the resulting value.

(* n1 n2...)

Multiples the numbers and returns the resulting value.

(/ n)

Returns 1/n.

(/ n1 n2...)

Divides n1 by each of the remaining numbers and returns the resulting value.

(QUOTIENT n1 n2...)

Divides the integer n1 by each of the remaining numbers and returns the resulting integer quotient. This function does integer division.

(REMAINDER n1 n2)

Divides the integer n1 by the integer n2 and returns the

remainder.

(MIN n1 n2...)

Returns the number with the minimum value.

(MAX n1 n2...)

Returns the number with the maximum value.

(SIN n)

    Returns the sine of the number.

(COS n)

    Returns the cosine of the number.

(TAN n)

    Returns the tangent of the number.

(ASIN n)

    Returns the arc-sine of the number.

(ACOS n)

    Returns the arc-cosine of the number.

(ATAN x)

    Returns the arc-tangent of x.

(ATAN y x)

    Returns the arc-tangent of y/x.

(EXP n)

    Returns e raised to the n.

(SQRT n)

    Returns the square root of n.

(EXPT n1 n2)

    Returns n1 raised to the n2 power.

(LOG n)

    Returns the natural logarithm of n.

NUMERIC COMPARISON FUNCTIONS

(< n1 n2...)
(= n1 n2...)
(> n1 n2...)
<<= n1 n2...)

(>= n1 n2...)

These functions compare numbers and return #t if the numbers match the predicate and #f otherwise. For instance, (< x y z) will return #t if x is less than y and y is less than z.

(LOGAND n1 n2...)

Returns the bitwise AND of the integer arguments.

(LOGIOR n1 n2...)

Returns the bitwise inclusive OR of the integer arguments.

(LOGXOR n1 n2...)

Returns the bitwise exclusive OR of the integer arguments.

(LOGNOT n)

Returns the bitwise complement of n.

STRING FUNCTIONS

(STRING-LENGTH str)

    Returns the length of the string.

(STRING-NULL? str)

    Returns #t if the string has a length of zero and #f
    otherwise.

(STRING-APPEND str1...)

    Returns the result of appending the string arguments.
    If no arguments are supplied, it returns the null
    string.

(STRING-REF str n)

    Returns the nth character in a string.

(SUBSTRING str start end)

    Returns the substring of str starting at start and
    ending at end (integers). The range is inclusive of
    start and exclusive of end.

STRING COMPARISON FUNCTIONS

(STRING<? str1 str2)
(STRING=? str1 str2)
(STRING>? str1 str2)
(STRING<=? str1 str2)
(STRING>=? str1 str2)

These functions compare strings and return #t if the strings match the predicate and #f otherwise. For instance, (STRING< x y) will return #t if x is less than y. Case is significant. #A does not match #a.

(STRING-CI<? str1 str2)
(STRING-CI=? str1 str2)
(STRING-CI>? str1 str2)
(STRING-CI<=? str1 str2)
(STRING-CI>=? str1 str2)

These functions compare strings and return #t if the strings match the predicate and #f otherwise. For instance, (STRING-CI< x y) will return #t if x is less than y. Case is not significant. #A matches #a.

CHARACTER COMPARISON FUNCTIONS

(CHAR<? ch1 ch2)
(CHAR=? ch1 ch2)
(CHAR>? ch1 ch2)
(CHAR<=? ch1 ch2)
(CHAR>=? ch1 ch2)

These functions compare characters and return #t if the characters match the predicate and #f otherwise. For instance, (CHAR< x y) will return #t if x is less than y. Case is significant. #A does not match #a.

(CHAR-CI<? ch1 ch2)
(CHAR-CI=? ch1 ch2)
(CHAR-CI>? ch1 ch2)
(CHAR-CI<=? ch1 ch2)
(CHAR-CI>=? ch1 ch2)

These functions compare characters and return #t if the characters match the predicate and #f otherwise. For instance, (CHAR-CI< x y) will return #t if x is less than y. Case is not significant. #A matchs #a.

INPUT/OUTPUT FUNCTIONS

(READ [port])

Reads an expression from the specified port. If no port is specified, the current input port is used. Returns the expression read or an object that satisfies the default-object? predicate if it reaches the end of file on the port.

(READ-CHAR [port])

Reads a character from the specified port. If no port is specified, the current input port is used. Returns the character read or an object that satisfies the default-object? predicate if it reaches the end of file on the port.

(READ-BYTE [port])

Reads a byte from the specified port. If no port is specified, the current input port is used. Returns the byte read or an object that satisfies the default-object? predicate if it reaches the end of file on the port.

(WRITE expr [port])
(PRIN1 expr [port])

Writes an expression to the specified port. If no port is specified, the current output port is used. The expression is written such that the READ function can read it back. This means that strings will be enclosed in quotes and characters will be printed with # notation.

(WRITE-CHAR ch [port])

Writes a character to the specified port. If no port is specified, the current output port is used.

(WRITE-BYTE ch [port])

Writes a byte to the specified port. If no port is specified, the current output port is used.

(DISPLAY expr [port])
(PRINC expr [port])

Writes an expression to the specified port. If no port is specified, the current output port is used. The expression is written without any quoting characters. No quotes will appear around strings and characters are written without the # notation.

(NEWLINE [port])

Starts a new line on the specified port. If no port is specified, the current output port is used.

OUTPUT CONTROL FUNCTIONS

(PRINT-BREADTH [n])

Controls the maximum number of elements of a list that will be printed. If n is an integer, the maximum number is set to n. If it is #f, the limit is set to infinity. This is the default. If n is omitted from the call, the current value is returned.

(PRINT-DEPTH [n])

Controls the maximum number of levels of a nested list that will be printed. If n is an integer, the maximum number is set to n. If it is #f, the limit is set to infinity. This is the default. If n is omitted from the call, the current value is returned.

FILE I/O FUNCTIONS

All four of the following OPEN functions take an optional argument to indicate that file I/O is to be done in binary mode. For binary files, this argument should be the symbol BINARY. For text files, the argument can be left out or the symbol TEXT can be supplied.

(OPEN-INPUT-FILE str ['binary])

Opens the file named by the string and returns an input port.

(OPEN-OUTPUT-FILE str ['binary])

Creates the file named by the string and returns an output port.

(OPEN-APPEND-FILE str ['binary])

Opens the file named by the string for appending returns an output port.

(OPEN-UPDATE-FILE str ['binary])

Opens the file named by the string for input and output and returns an input/output port.

(GET-FILE-POSITION port)

Returns the current file position as an offset in bytes from the beginning of the file.

(SET-FILE-POSITION! port offset whence)

Sets the current file position as an offset in bytes

from the beginning of the file (when whence equals 0), the current file position (when whence equals 1) or the end of the file (when whence equals 2). Returns the new file position as an offset from the start of the file.

(CLOSE-PORT port)

Closes any port.

(CLOSE-INPUT-PORT port)

Closes an input port.

(CLOSE-OUTPUT-PORT port)

Closes an output port.

(CALL-WITH-INPUT-FILE str proc)

Open the file whose name is specifed by str and call proc passing the resulting input port as an argument. When proc returns, close the file and return the value returned by proc as the result.

(CALL-WITH-OUTPUT-FILE str proc)

Create the file whose name is specifed by str and call proc passing the resulting output port as an argument. When proc returns, close the file and return the value returned by proc as the result.

(CURRENT-INPUT-PORT)

Returns the current input port.

(CURRENT-OUTPUT-PORT)

Returns the current output port.

CONTROL FEATURES

(EVAL expr)

Evaluate the expression in the global environment and return its value.

(APPLY proc args)

Apply the procedure to the list of arguments and return the result.

(MAP proc list...)

Apply the procedure to argument lists formed by taking corresponding elements from each list. Form a list from the resulting values and return that list as the result of the MAP call.

(FOR-EACH fun list...)

Apply the procedure to argument lists formed by taking corresponding elements from each list. The values returned by the procedure applications are discarded. The value returned by FOR-EACH is unspecified.

(CALL-WITH-CURRENT-CONTINUATION proc)
(CALL/CC proc)

Form an "escape procedure" from the current continuation and pass it as an argument to proc. Calling the escape procedure with a single argument will cause that argument to be passed to the continuation that was in effect when the CALL-WITH-CURRENT-CONTINUATION procedure was called.

ENVIRONMENT FUNCTIONS

(THE-ENVIRONMENT)

Returns the current environment.

(PROCEDURE-ENVIRONMENT proc)

Returns the environment from a procedure closure.

(ENVIRONMENT-BINDINGS env)

Returns an association list corresponding to the top most frame of the specified environment.

(ENVIRONMENT-PARENT env)

Returns the parent environment of the specified environment.

UTILITY FUNCTIONS

(LOAD str)

Read and evaluate each expression from the specified file.

(LOAD-NOISILY str)

Read and evaluate each expression from the specified file and print the results to the current output port.

(TRANSCRIPT-ON str)

Opens a transcript file with the specified name and begins logging the interactive session to that file.

(TRANSCRIPT-OFF)

Closes the current transcript file.

(EXIT)

Exits from XScheme back to the operating system.

(GC [ni vi])

Invokes the garbage collector and returns information on memory usage. If ni and vi are specified, they must be integers. Node and vector space are expanded by those amounts respectively and no garbage collection is triggered. GC returns an array of six values: the number of calls to the garbage collector, the total number of nodes, the current number of free nodes, the number of node segments, the number of vector segments and the total number of bytes allocated to the heap.

(RESET)

Returns to the top level read/eval/print loop.

SYSTEM FUNCTIONS

(%CAR pair)
(%CDR pair)
(%SET-CAR! pair expr)
(%SET-CDR! pair expr)
(%VECTOR-LENGTH vect)
(%VECTOR-REF vect n)
(%VECTOR-SET! vect n expr)

These functions do the same as their counterparts without the leading '%' character. The difference is that they don't check the type of their first argument. This makes it possible to examine data structures that have the same internal representation as pairs and vectors. It is *very* dangerous to modify objects using

these functions and there is no guarantee that future releases of XScheme will represent objects in the same way that the current version does.

This version of XScheme uses the following object representations:

Closures are represented as pairs. The car of the pair is the compiled function and the cdr of the pair is the saved environment.

Compiled functions are represented as vectors. The element at offset 0 is the bytecode string. The element at offset 1 is the function name. The

element at offset 2 is a list of names of the function arguments. The elements at offsets above 2 are the literals refered to by the compiled bytecodes.

Environments are represented as lists of vectors. Each vector is an environment frame. The element at offset 0 is a list of the symbols that are bound in that frame. The symbol values start at offset 1.

Objects are represented as vectors. The element at offset 0 is the class of the object. The remaining elements are the object's instance variable values.